Research report 137

ESSENTIAL AND EPHEMERAL KNOWLEDGE

The Culture and Education of Computer Specialists

David Park

(RR137)

Text of an Inaugural Lecture, given at Warwick University, on 28 November 1983.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

November 1983

ESSENTIAL AND EPHEMERAL KNOWLEDGE

The Culture and Education of Computer Specialists

Text of an Inaugural Lecture, given at Warwick University, on 28th November 1983

David Park

Department of Computer Science
Warwick University
Coventry CV4 7AL

In memory of my father and of my brother
James Ritchie Park (1902-1952)
John Gavin Park (1931-1983)

ESSENTIAL AND EPHEMERAL KNOWLEDGE

The Culture and Education of Computer Specialists

Text of an Inaugural Lecture, given at Warwick University, on 28th November 1983

David Park

Department of Computer Science
Warwick University
Coventry CV4 7AL

1. Introduction.

There is a short story by E M Forster which captures a fear that many of us must have about computers. Called "The Machine Stops", it was published in 1928, long in advance of modern computers. In it, Forster imagines a world controlled by a computer (he calls it a 'machine') which meets a catastrophic end. Apparently the computer has begun to fail. And the 'Mending Machine', the computer designed to fix all faults, is also failing; and noone is able to fix the Mending Machine. So the computer, and the world that now depends on it, collapse catastrophically.

This should be a nightmare shared by many computer people, just because the dynamic, once one accepts its science fiction setting, is so plausible. It is a commonplace to come across gadgets with errors which we cannot fix, because we do not have the information, and maybe the parts, needed to fix them. The manufacturer can do so, but is inaccessible. When the gadget is a computer system, however, there may be added reasons for the obscurity. If the system is one which administers some resource, we might use any knowledge of it to gain unfair advantage for ourselves. There is a security problem. We might commit a computer fraud, or gain access to other secret information. There is also, more probably, a commercial calculation. The information we need would enable us to copy (or modify) the system, and would infringe the manufacturer's proprietary rights, which he is not otherwise able to protect. There is a proprietary problem. We might pirate the system, or encroach on a lucrative service business. Besides, we might introduce errors in any modification, and pass them on to other users. In either of these cases, of course, we can no doubt obtain the manufacturer's service (though at a price set by him). The security and proprietary problems give cause for concern, and perhaps cause for legislation. It is important that the market for computer systems operates equitably. But what is most worrying is the most plausible cause of Forster's disaster. There was no record of the information that was needed. It had been lost, or was never recorded. And the one man who knew anything was now dead.

2. Culture.

Technical information is of supreme importance to the computer industry. It can be claimed as the prime source of value within it. If one is willing to accept machine readable programs and etchings on silicon as "information", this is clear. Even without silicon, it is now arguable. Whatever the medium, the costs of design are immense. In the realm of software, the fact that the user does not always see this cost is a vagary of the market. Programs are not easily protected by copyright. The market price is related to the cost of evading the current state of copyright law, rather than to the effort of the designer. As long as this results in a discrepancy, commercial organizations will try to put tight commercial secrecy on the detailed design of any software they produce. Remembering the Forster fable, this is one reason to look to the public sector to produce or commission software of public consequence.

Very complex information is involved. A community of systems designers can be looked at as proceeding by a sequence of operations on its **information base**. The system designer uses this base as a source of useful utilities and information, studies information relevant to his problem of the moment, then uses his understanding and the information base, to develop (or modify) a computer system. If his system is one likely to be useful to other designers, technical information about it is added to (or modified in) the information base. If this happens, the system becomes "software" so far as the community is concerned. The intellectual effort needed in this process, together with the entrepreneurial talents needed to direct this effort, seem to be prime limiting factors in the development of the computer profession and thereby of the industry.

The use of a programming language is the standard means for specifying some of the technical information desirable. If it gets to be software, it needs to be augmented, at least, by **documentation** which tells the user precisely how he is to use the program and what to expect. It may also need installation instructions, and maybe maintenance information in addition to program.

Uncompiled programs (as far as they are available) have a natural hierarchical structure -- program A is written in language B which is implemented in language C which is implemented in .... -- and there may be anything up to 5 or 6 levels in this hierarchy. The "implementation" of each language is a "compiler" or "interpreter" which is an item of software in the information base. The last implementation is one in machine code, to be run on hardware.

The texts of programs must be intelligible enough to allow other systems programmers to correct errors or modify them, particularly if they are to become software. Intelligibility disappears when the program has been compiled, when there is sufficient information to run the program, but rarely enough to comprehend it. This fact permits wide distribution of a program along with comparative (but not absolute) secrecy as to its working.

The profession appears to have come a long way from the managerial and technological vacuum which caused the **software crisis** of the 1960s, when huge resources were absorbed by software projects which proved far more difficult than anticipated. The difference has been due, in part, to the developments of **software engineering**, which concerned itself with organizing large software projects, which developed appropriate **software tools** for the purpose, and improved standards of programming and documentation. But perhaps most of all the improvements arose from improved communication among programmers, through the multi-access computer systems which followed CTSS, developed at MIT in the early 1960s, and through the networks which followed ARPANET, developed in the USA in the early 1970s, which was to link centres of Artificial Intelligence (AI) research there.

The usage of both projects surprised their designers. CTSS was intended as a source of cheap shared computing power. Its designers were surprised by the extent to which it was used for work which had no "computing" component at all. It was heavily used, it turned out, for text processing. Many technical papers were produced on it. Its (fairly primitive) facilities were an improvement on hard-pressed MIT typists, and word-processing system functions were available at an early stage. They were then used heavily, in conjunction with the disc filing system which held texts that were in process of being worked on. This is essentially what we now call word-processing.

ARPANET was intended to permit AI researchers to run large AI programs remotely, at installations with the heavy computing power needed for them. Actually the traffic between installations was dominated by text again, of what we now call **electronic mail** -- messages between individuals.

There is no doubt that the existence of multi-access computers and networking has affected communication between computer people in a profound way. Facilities very much like these are already affecting routine office-work, as we all know. Simple improvements -- the freedom from paper as a medium, and from synchronised communication, as needed using conventional mail and telephone -- have had far-reaching results. It is not quite so well known that they can also affect education. At Warwick we now run lecture courses which can be administered almost entirely through the computer, since both the lecturer and the students have easy access to terminals. Since it is a university regulation, spoken lectures are given. However the lecturer's notes are available for inspection through terminals. Student comments come (apparently more readily) through electronic mail, and the notes can be amended as a result. Example sheets are distributed through the system, and answers returned through mail. "Corrected" example sheets (their answers, with lecturer's comments edited in) are returned, again through mail. No paper is used. All communication is throught the computer system. Apparently the only function that could not be performed satisfactorily in this way is the final examination -- for security reasons. [The security of a university installation gets extremely well tested by its students!] It is to be hoped that schools will soon see similar developments.

Bringing all this communication to the computer specialist's terminal means that he develops a very special relationship with "the system", since almost every aspect of his career now turns on his interactions with it, and with his colleagues through it. Social chat, professional gossip, office paperwork, professional papers, program writing and debugging, program documentation, appointments, conference announcements, even political manifestos, all pass through the system, and can be dealt with if and when he wants it. "The system" comprises a whole self-contained, always changing world for him, whose detail he can never hope to master completely. His whole career turns on how he chooses to explore it, use it, and add to it. As well as an information base, the system is a medium for communication for him. It is at the same time his library and his filing system and his secretary -- and the distinctions between these functions become blurred. He consults the system, modifies it, sends mail and receives mail -- all when he wishes. He can incorporate any information he has access to in what he sends, and he can modify it in any way he feels appropriate. He can form a special interest group or "subscribe" to someone else's to whom mail can be broadcast -- and he can resign and form another if this results in too much "junk mail", and so on.

The most important effect of network facilities is to create a much larger work-community, structured in a way which need not depend on geographical location. This allows very specialised communities to form, with research and development taking place in a thoroughly distributed manner. It also allows access to expertise which one could not expect to be available locally, including the designers of the systems available as software over the network. This permits evolution of software which is otherwise difficult to arrange. The designer can submit first his specification, then preliminary versions of his system (possibly just to selected individuals), can invite comments and offer tuition, and can modify his product in the light of the criticism, experience and interest (or lack of it) all of which is very quickly obtained through the network. He can also enroll the assistance of the community in finding applications for his product. He can look at whatever rival products are available in the network, in as much detail as its designers will let him. The result is an enviable process which, depending on the community, ensures wide criticism and testing of a product while its designer is in a position to respond easily.

An example of an impressive software product, which probably could not have been developed at all without this sort of interaction, is the MACSYMA system, based at MIT, for automating algebraic manipulation. MACSYSMA has turned into an extremely valuable tool for applied mathematicians who need to manipulate and simplify large mathematical expressions -- a familiar and daunting situation for them.

To summarise, the new facilities have created new work communities, more favourable to the production of software and other computer-related development. Quick, wide and selective communication is critical. What is communicated, and how efficient it is, are other questions, of course. The US ARPA network has proved so successful that one hears complaints from British computer scientists, that they have lost social and professional touch with some of their colleagues, who respond only to what happens through their computer terminals. US scientists, incidentally, frequently use work terminals from their homes -- partly because the necessary communication equipment has always been cheaper in the US, but also through the accident that local telephone calls in the US are often not charged individually. One wonders to what extent the migration of Britain's AI talent was due to the carrot of ARPANET and the large interested community it created, rather than to the stick of SRC's Lighthill Report.

Some facts of the computer specialist's life have not changed. There is still the necessity to debug one's designs. For some otherwise talented people, the trauma of discovering that their programs contain errors puts them off programming permanently. For others, debugging is a species of intellectual sport with many satisfactions. Certainly the process requires the exercise of intelligence and thoroughness. And, on a large software project, there will still be bugs remaining which go undetected, however gifted the programmers. Fallibility of system designers is an unpalatable but real fact of life. Its realisation affects computer people in ways we more usually associate with a sense of original sin. The best practical answer we know is to allow software to evolve in environments where bugs can do minimal harm, and can quickly be corrected. There are more radical proposals, that programmers be required to prove correctness of their programs -- though these have still to find complete favour with programmers.

3. Ephemeral Knowledge.

The community described so far developed around the big shared "mainframe" systems common from the late 1960s to the present day. Its research and other preoccupations gave rise to the beginnings of academic computer science -- with ideas principally coming from the large US community, but with a very important leavening of ideas from Europe. I will try to indicate the principal lines in a moment. But first I want to digress.

What I have described may seem strange to those who come from the newer computer culture which is associated with the microprocessor. They would not see themselves as forming such a coherent community. The new microtechnology has, obviously, enlivened work in computer science, since it has markedly moved the boundaries of what is practicable. In the Japanese Fifth Generation studies and in the British government's Alvey report, we see the economic priority that has been assigned to progress in these fields. Nevertheless there is an indirect effect on the education system which is more worrying. It seems to be an effect common to innovations which are economically sensational -- as the microchip indeed is.

When I last gave a public lecture here, 10 years ago, about computer science, I compared the mood that had produced it with that in the engineering industry in Britain in the great age of steam, between, say 1830 and the Great Exhibition of 1851.

> "There was buoyant optimism, great ingenuity, and a feeling of great self-sufficiency -- that all that was needed was a knowledge of developing craft tradition, and the simple idea - that if it moved you could mechanise it, with "live steam"...."

Actually, there are other interesting aspects of the analogy. My remarks on the supreme importance of technical information, for example, has an interesting similarity to Marx's Labour Theory of Value -- peculiarly appropriate to the transition from mechanical power to information-processing power. [And one can envisage a development of the theme. Power, in some organizations we know, is obtained by the careful acquisition of information about it, and is maintained by just as carefully denying information about it, and so on.]

More seriously, one is concerned at the effect of this mood on educational values. While fortunes are easily made by keeping up with microtechnology, and picking off applications from an apparently endless train of opportunities, it is tempting to want education, if any, to be geared solely to vocational training for that role, or for whatever else is so immediately profitable. Speculative research is squeezed out. The investment in high quality university personnel is not made. An education which is more than adequate for the vocation is transformed into one which is only adequate for that vocation (but which, perhaps, gives its recipient a head start in the race for profit).

What seems to have happened with 19th century engineering is this -- that Britain's success with mechanization was so extreme that education was conceived solely as apprenticeship, and was not seen as otherwise essential to engineers. Many apprenticeships actually gave their recipients a high-grade and imaginative training. Nevertheless, the communities of engineers that were formed, based on education and research, in France, Germany and the USA, in reaction to the British success, did not form in Britain until much later, when the best of the early British engineers who might have led them were already dead. The consequences to the British engineering profession appear to be still with us.

My fear is that the same dynamic threatens to operate in computer science today. The pressures to confine ourselves to the vocational and to concentrate on immediate applicability in our research, are very real, and threaten to deprive us of the basic research which we need. For it is around this research and from the students who are able to interest themselves in it that a growing corps of teaching and research expertise can be expected to develop. That corps of expertise is essential if we are to serve the educational and research needs of the future in our subject -- needs which will be immense.

It should be said that the government's Alvey and IT initiatives appear to have intentions which offset this fear, in that they wish to support a certain amount of basic computer science research, which is seen as relevant to Fifth Generation computer developments. It remains to be seen whether this will produce the general support for the subject that it really needs.

It is the pressure for vocational education which produces the conflict between the **ephemeral** and the **essential** which I refer to in my title. There is a mass of **ephemeral** knowledge which could be learnt, and which would fit students for very particular jobs in the industry as it is now. Moreover, the ephemeral knowledge is

complex, abstract and presents a variety of intellectual challenge all of its own. Nevertheless, I claim, the proper way to acquire such ephemeral knowledge is piecemeal and on the job, in the style of my ideal systems designer, who is seen as absorbing just such information from his information base, and continuing to do so for the foreseeable future. The role of academic computer science is to fit him to absorb such material well, rather than to preempt the absorbing process.

Ephemeral knowledge is speculative, in the commercial sense. The rewards of possessing it are not predictable. They may be immense or empty, depending not on absolute criteria, but rather on the vagaries of some market. It is difficult to fit such knowledge into the criteria usually adopted by academics for judging research -- who take the notion of "contribution to knowledge" as an absolute which can be judged in isolation. The bulk of technical information about computing, which is immense, is therefore not "knowledge" in the academic sense.

4. The Essential Thread.

I should return now, briefly, to the thread of computer science ideas which can be counted as a part of the academic subject, and which arose from the culture of the computer specialist over the last 25 years. I should make it quite clear, first of all, that I am discussing only those aspects of computer science education which are not obviously vocational -- which have arisen from the thoughts and arguments of computer people, rather than from their practice. I have chosen to organise my material here in what may seem a haphazard way, which springs more from personal history than from the categories we would now impose for the sake of curricula etc.

The first individual I want to discuss is **John McCarthy**, who made important early contributions to the field, as well as persuading me and others at MIT in the late 1950s that there was a very real intellectual challenge in it. His contribution was immense, though it was to some extent incidental to his main interest, which was AI. McCarthy wanted to write very ambitious computer programs, and followed through the implications of this ambition absolutely thoroughly. This meant facing up squarely to the methodology of computing, which at that time was still very primitive. His ideas about AI led him to develop a connection with mathematical logic; he proceeded to try to formulate it. The result was a scheme called the "Advice Taker", a system which could take **advice** about the world in the form of statements in a system of formal logic, and take assertions about needs in the same form. It would then use this information by drawing conclusions from its assertions, formally, according to the rules of symbolic logic. The innovation was to take seriously **common sense** reasoning -- for example solving a simple task, of getting from his desk to the airport, on the basis of knowledge about his house, car, geography etc. Previously formal logic had been considered mainly in contexts which called for carefully reasoned argument, for example in mathematics. This idea made reasoning into a sort of symbol- processing, for which an appropriate programming language was needed. The idea of **list-processing** for this purpose was not new, but McCarthy's formulation of his list-processing language LISP was exceptionally elegant. McCarthy saw it desirable that programs should themselves be objects that could be manipulated by other programs. One reason for this might be that reasoning by the Advice Taker would ultimately involve reasoning about actions, and that actions could be compounded into programs. So there should be reasoning about programs, and programs should be well-structured enough for this to be possible. The only concepts that were really well-understood as regards formal reasoning were those of mathematics, concepts such as those of sets, functions and relations, particularly those familiar in modern algebra. This made McCarthy wish to formulate programs in the style of formal mathematics. The result was LISP, which is still, in various dialects, a principal standard language of AI, and also a source of much inspiration to theoreticians, since it began to make reasoning about programs non-mysterious, and programming itself into a most abstract business. With LISP, the computing machinery began to vanish into the background. What was left was clearly nearer to the "essence" of computation, a system that could be comprehended in purely non-mechanistic terms.

Functional programs in the style of "pure LISP" can conceivably be run with a great deal of parallelism -- so **functional languages** and machine architectures derived from them, are now much studied. The connection with logic recently gave rise to a most interesting suggestion, derived from a device originally discovered in logic as long ago as 1920 -- giving rise to a class of **combinator machines**, running functional programs efficiently with a very simple instruction code, since the device eliminates the need for **variables**. Another observation (lazy evaluation) has extended these languages to operate on infinite objects, e.g. infinite sequences -- making computationally respectable what has always been accepted in mathematics.

Direct descendents of the Advice Taker, which itself was never completed, are the **expert systems** which are now capable of making medical diagnoses on the basis of clinical evidence, and of knowledge acquired from human

experts, in a form very similar to that envisaged by McCarthy for "advice".

The interest generated in automatic theorem-proving as a technique for AI led to yet another view of computation, by **Kowalski**, then working at Edinburgh. He looked at computation as itself a variety of logic reasoning. This led to the formulation of the language PROLOG, which has been adopted by the Japanese as a basis for fifth generation computer design, with expert-like systems as the main intended application, and with vast investment of resources.

It was McCarthy who coined the term **Theory of Computation** for his interests in proofs about programs. These quickly attracted computer science interest, as the basis for alternative methodologies, which would reduce the dependence on program testing as a means of finding errors.

It was John McCarthy and the British computer scientist **Christopher Strachey**, then at Cambridge, who suggested the possibility that computers be used simultaneously by many people, giving rise to the first conceptions of multi-access systems such as CTSS. Strachey himself had a considerable influence on the development of the theory of computation initiated by McCarthy. Like McCarthy, he was a man of extremely independent temperament, not to be put down by professional humbug and current fashion among his colleagues, most of whom at this time tended to consider conceptual difficulties as quibbles compared with the difficulties of getting new technology to work efficiently. Strachey stuck to his view: that we did not yet know enough about the essential structure of programming to be able to program well. He pioneered the study of the **semantics of programming languages** -- first in collaboration with **Peter Landin**, who had absorbed the LISP ideas from McCarthy. Landin investigated the connection with the mathematical **lambda-calculus** , and came to describe other programming languages as "syntactically sugared" variants of this more fundamental system. Strachey interested the logician **Dana Scott** in his problems. Scott was to formulate a **theory of domains**, essentially a variant of the constructions of classical set-theory, as a foundation for computation theory. This turned out to be a brilliant device, with much mathematical, as well as computational, interest. It is now widely accepted as a technique for describing the structures and processes of computation. He, Strachey and their students proceeded to apply the new theory to descriptions of many languages, which were elegant and could be used to formalise proofs about programs. **Robin Milner**, now at Edinburgh, devised a computer system LCF, to check, and develop general methods for generating, proofs based on Scott's theory. Among many interesting developments from this, **Michael Gordon**, at Cambridge, has used LCF to develop systematic ways of proving correctness of hardware, with the intention of providing VLSI designers with realistic development tools, in which they can establish correctness of their designs before testing.

Strachey's interest in contemplating alternatives to conventional programming led him into research on operating systems, insisting on programming them in as abstract and intelligible a way as possible. His work in this area may have influenced the design of the UNIX operating system at Bell Laboratories. UNIX was another example of a successful product originating in an insistence on exploring an unconventional but natural abstract model, rather than from practical motives. UNIX exploits the use of a versatile high-level language as the command language for the system, and operating on **streams**, ground pioneered by Strachey and Stoy.

Complexity theory was pioneered and popularised by **Donald Knuth**, operating from Stanford, whose 3-volume "Art of Computer Programming" contains a multitude of complexity results. Given a computing procedure p(n) with an integer input parameter n, the **time complexity** of p is the function of n which is the least time taken to compute p(n). The time complexity for good sorting procedures, to sort n numbers, is well-known to be "of order nlogn", i.e. a function which behaves like nlogn (to within a multiplicative constant) for large values of n. This is a typical elementary copmplexity result.

Knuth used a host of combinatorial methods to analyze the complexity of various procedures, concentrating, of course, on finding algorithms which established that the complexity was small. The field was given much stimulus by a connection made with the computability theory of logicians -- a connection made by **Steve Cook**, now at Toronto. This has resulted in a mass of work in the area, particularly on a problem which is still open. Some computer scientists claim it to be the most important mathematical problem of the moment:

Call a problem p(n) **polynomial computable** if has a time complexity of order bounded by n^k for some k. There is a class NPC of problems (the **nondeterministic polynomial complete** problems) such that

(a) NPC contains many problems of practical interest
(b) either every problem in NPC is polynomial computable or none is,
and

(c) we do not know which alternative of (b) is the case.

The positive result, that all of NPC is polynomial computable (that P=NP, in Cook's terminology) would be sensational, in view of (a). It appears more likely that the reverse is true, that all of NPC are computable, at best, with time complexity exponential in n. This would make them infeasible for even small values of n (recall the exponential growth obtained by doubling up grains of rice on a chessboard -- the number of grains on the 64th square is 2^{63}, about 8,000,000,000,000,000,000.)

Complexity theory and related topics have interesting applications to the theory of cryptography, and to various schemes for ensuring secure communication in computer networks.

Structured programming was pioneered by **Edsger Dijkstra**, at Eindhoven, and by **Tony Hoare**, now at Oxford. It had a marked influence on standards of programming style and on the **design of programming languages** , since it dealt with elementary reasoning about programs written in more or less conventional notation, and with the constraints on the language and style which might guarantee correctness. The language PASCAL was given a shape which permitted these rules to be applied easily, and PASCAL-like languages have continued to emerge since.

Dijkstra is an articulate, sophisticated programmer, whose works are full of careful analysis of different ways of thinking about programming problems and about the associated problems of designing languages for them.

Hoare has extended his ideas particularly to the study of concurrent systems, in common with a preoccupation of Milner at Edinburgh. Collaboration between Hoare and **David May**, of INMOS, has resulted in a most interesting development, of a language OCCAM (a precursor was EPL, designed and implemented by May at Warwick) to be used as the code for a new class of microchip component, the **transputer**, a chip which combines memory, processing and communications facilities, intended for use in combinations to form collaborative computing networks with large numbers of nodes. Development of such networks, and the methodology of programming to go with them, is likely to be an important component of the British contribution to Fifth Generation computer design.

5. Conclusion.

I have attempted, by reviewing the work of a few individuals, to give an impression of basic research in computer science. The list is by no means complete. What is remarkable is the number of practical developments that have only occurred after stimulus from individuals such as these. These computer scientists had the vision and breadth of imagination to push through projects which, while complex in their implementation, corresponded to natural and coherent theoretical constructs. With contributions such as these, computer science becomes an independent academic subject, with an ethos which is, to some extent, independent of patronage or application. These are, I believe, the latest of a long succession of worthy contributors to rational argument and analysis. Their subject is to be taken seriously, and not only for reasons of technological or academic expediency -- reasons which may otherwise blind people to genuine intellectual achievement.